

A Generalized Streaming Model For Concurrent Computing*

Yibing Wang[†]

Abstract- Multicore parallel programming has some very difficult problems such as deadlocks during synchronizations and race conditions brought by concurrency. Added to the difficulty is the lack of a simple, well-accepted computing model for multicore architectures—because of that it is hard to develop powerful programming environments and debugging tools. To tackle the challenges, we promote a generalized stream computing model, inspired by previous researches on stream computing, that unifies parallelization strategies for programming language design, compiler design and operating system design. Our model provides a high-level abstraction in designing language constructs to convey concepts of concurrent operations, in organizing a program’s runtime layout for parallel execution, and in scheduling concurrent instruction blocks through runtime and/or operating systems. In this paper, we give a high-level description of the proposed model: we define the foundation of the model, show its simplicity through algebraic/-computational operation analysis, illustrate a programming framework enabled by the model, and demonstrate its potential through powerful design options for programming languages, compilers and operating systems.

1 Introduction

1.1 Motivation

When searching for a general approach to the challenges [2] faced by multicore programming, particularly deadlocks during synchronizations and race conditions [47] introduced by concurrency, which render solutions built on conventional multithreading models [13, 52, 68] unappealing to programmers, we found that there was a need for a unified computing model on which computer professionals at programming language design level, compiler design level, and operating system design level could benefit from each others’ works; and a generalization of the stream computing model [12] seemed to have the potential to serve as such a design model for future generation computing. Some reasons are explained as follows.

Information processing on modern computers has never followed a monotonous path, but the mainstream has come along a relatively unified way due to the Turing machine abstraction in theory [26] and von Neumann model [3] in design. The adoption of multicore architectures, however, brings the industry into a new chaos at various computing levels ranging from programming language designs to operating system designs. Not only are programmers often frustrated with primitive solutions or the lack of ideal software tools (e.g. programming languages and debuggers), but also are chip makers facing challenges in delivering new designs [34, 48]; whereas tools’ builders are constrained by the multithreading model, such as in the cases of OpenMP [52] and C++-0x thread library extension [68], or by imperative languages, such as in the cases of CUDA [49, 51] and OpenCL [31]. To change the impasse situation where researchers and developers at different computing levels have uncertain expectations from each other, first a well-accepted computing model is needed, i.e. we need a model that can guide designs at different computing levels.

PRAM [19] has often been used as a conceptual parallel computing model in demonstrating synchronized executions such as concurrent read concurrent write and concurrent read exclusive write. We argue that PRAM captures certain essences such as parallelism in concurrent computing¹ but not other substances such as precedence constraints in operation; and the very nature of its dependence on shared memory limits PRAM’s power in design. Therefore, if PRAM is only a generalization (or parallelization) of the RAM [26] model (Turing machine equivalent) for *abstract analysis*, what is the parallel-formed generalization of the von Neumann model and its alternatives for *design*? Conventional threads and transactional memory [11, 24, 35] are not the answer; they are implementation details with little extension to the von Neumann model. Many people

*First draft: Jan. 31, 2010; this draft: Dec. 07, 2010.

[†]Author contact: williamwang@acm.org.

¹In this paper, *concurrent* and *parallel* are interchangeable in most cases, though we tend to regard concurrent as having broader meaning in computing than parallel does. The authors of [16] think parallel systems are mostly synchronous, whereas concurrent systems also include distributed cases that are mostly asynchronous.

agree that threads are low-level programming utilities and should remain so; transactional memory may not be computationally productive nor energy-efficient when transactions are large in size or long in time [24].

Stream computing [12, 17, 33, 44] has been around for more than a decade. Perhaps due to their humble origin (i.e. graphics processors; not their noble originators) in the computer industry, technologies based on stream computing were often used only as performance accelerators. While surveying the landscape of multicore parallel computing, we realized that, maybe, the application of stream computing model should go beyond even GPGPU computing, and a generalization of the model could be applied to general purpose software designs and programming as well. Note our pursuit of a general design model was not to find a killer solution (i.e. the implementation of a certain method) for all applications, but to find a design principle in handling some of the most difficult yet fundamental problems, such as parallelization and nondeterminism.

To serve our purpose, we recommend the separation of computing models for *abstract analysis* from computing models for *generalized design*. In [57], Snyder expressed similar idea but from a different angle; he emphasized that a useful parallel machine model should be able to capture key features that have impact on performance, which to us is a design (model) issue. Based on such understanding, plus other people’s publications [10, 11, 20, 25, 35, 36, 37, 40] and our own experiences², we propose a generalized stream computing (hereinafter known as GeneSC, pronounced “g-e-n-e-s-i-s”) model as a guideline for multicore computing. Our expectations of the model can be summarized as follows.

- It will help with defining language constructs for expressing concurrency structures in programs and algorithms, but leaving parallelization and synchronization controls to compilers, runtimes and/or operating systems.
- It will bring parallelizing compilers out of the confinement of sequential execution constructs such as loop nests.
- It will bring in operating system designs for tractable, reliable and concurrent instruction sets executions.
- It will support backward compatible programming schemes, be it structured, object-oriented,

or functional; and the re-use of existing (modified when necessary) library routines.

- It can be reduced to sequential case for debugging purpose or when used for single core architectures.

Furthermore, we regard algorithm design as an inseparable part of achieving high-performance in multicore parallel computing. We believe that, due to different algorithmic thinking, solutions to an application problem may show different concurrency structures, which have different levels of difficulty when mapped to the underlying runtime environment and hardware. By supplying an expressive computing model, we give programmers the needed power in defining the least convoluted concurrency structures in their solutions. We use the following example to further justify our motivation.

1.2 Example

A naive solution to the N -body problem [5, 66] has the computation complexity of $O(N^2)$. Figure 1 shows such an algorithm, where `gforce()` is a function for calculating gravitational forces from the other $N - 1$ objects to the j th object, following the Newtonian laws of physics. To parallelize this algorithm, we often rely on loop nest parallelization by hand or through compilers [1, 18].

```

1 /* calculate gravitational forces in time
2    sequence from zero to tmax */
3 for (ti = 0; ti < tmax; ti++)
4 {
5     for (j = 0; j < N; j++)
6     {
7         f = gforce(j);
8         vnew[j] = v[j] + f * dt;
9         xnew[j] = x[j] + vnew[j] * dt;
10    }
11    for (j = 0; j < N; j++)
12    {
13        v[j] = vnew[j];
14        x[j] = xnew[j];
15    }
16 }
```

Figure 1: The N -body algorithm of complexity $O(N^2)$.

Instead of doing $O(N^2)$ direct force integrations, an improved design [5] recursively divides the space into smaller cubic cells until each cubic cell contains no or only one of the N objects, then builds a tree with cubic cells that have more than one objects as the intermediate nodes and cells with only one object as the leaves. Gravitational forces among the N objects are calculated in a way where far away objects

²For example, the author wrote his first multithreaded server in 1997. Part of that early experience was published in [64].

are approximated by their conglomeration masses at their geometry centers. Figure 2 shows the stream-like skeleton of such an algorithm with complexity $O(N \log^N)$, where each inner-loop function consists data parallelism.

```

1 for (ti = 0; ti < tmax; ti++)
2 {
3     space_subdivision();
4     tree_construction();
5     mass_center_calc();
6     approximate_force();
7     position_update();
8 }

```

Figure 2: An improved N -body algorithm of complexity $O(N \log^N)$.

The N -body problem is interesting in three ways. First, it is a large computation problem that must be solved on parallel computers. Second, inter-node communication (i.e. synchronization) overhead in executions of each parallelized inner functions is not negligible. Third, it has the potential to show how languages (i.e. the expression of reasoning) may affect algorithm designs.

Further improved algorithms such as [4, 65] aim to reduce communication overhead. Multicore processors may reduce the overhead even more through shared memory. Besides hardware support, a suitable computing model is needed. The GeneSC model that we propose has considered such applications as well as many others.

1.3 Organization

The rest of the paper explains what is this generalized stream computing and what are the enabling technologies. Section 2 first highlights a few important concepts in the problem domain of interests and then gives a definition of the GeneSC model, followed by Section 3 on algebraic/computational operation analyses, Section 4 on a programming framework enabled by our model, and Section 5 on design considerations including three crucial additions that depart from traditional practices. Section 6 introduces one potential application. Section 7 discusses related works. Section 8 gives the concluding remarks.

2 Definition

We use synchronism and asynchronism for discussing operations' timing or ordering characteristics. We use determinism and non-determinism for discussing

bounded program behaviors, with non-determinism refers to program behaviors that are different (but well-defined) from run to run even for the same given inputs; and indeterminacy for unbounded, i.e. unpredictable program behaviors. Synchronous operations without randomized functions normally yield deterministic results; asynchronous operations may yield either deterministic (when no ordering is needed) or non-deterministic (when ordering is required but, by mistake, not enforced) results. Shared memory could add more complexities, e.g. when supposedly deterministic operations show non-deterministic or even indeterminacy behaviors, or when non-deterministic operations show indeterminacy behaviors, due to race conditions.

Deadlocks and race conditions are notorious because they bring indeterminacy. General races [47] can be defined as concurrent accesses to shared data with at least one access is a write operation. General races may be acceptable if they just cause non-determinism, but are unacceptable if they cause indeterminacy. Data races are special cases of general races; data races are program bugs, e.g. missing shared locks in concurrent accesses to shared data.

Applications show parallelism in two ways: data parallelism and task parallelism (we consider pipelined concurrency as a subtype of task parallelism). Performance in data parallelism cases relies on data processing speed, whereas performance in task parallelism cases relies on overall throughput.

To make a parallel program work correctly, we prevent indeterminacy; to make the behavior of a parallel program tractable, we deal with non-determinism.

In our GeneSC model, programming, compilation and runtime designs adopt a concept that views computing as applying well-defined functions to flows of data sets, which we call *stream data*. At the core of the model is an entity, $E = (k, d, r)$, that encapsulates an execution unit consisting of three elements:

- k : a close-form function
- d : well-defined input/output data
- r : relations with other entities

The close-form function is in fact a non-trivial private algorithm and is often called a (mathematical) kernel. The input/output data defines a function's interactions with the outside, without exposing intermediate computation results. Inter-stream-entity relations offer timing/ordering dependence.

This model is different from conventional multi-threading models in at least two ways: First, stream

entities are functional units, whereas threads are execution units. Second, explicit concurrency structures defined by stream entity's syntax and semantics allow optimization in compilation and scheduling at runtime, whereas threads prescribe execution structures without rich timing/ordering knowledge.

Some people might suggest that a stream entity resembles a neuron in artificial neuron networks, except that the former normally has a linear function whereas the latter a non-linear function. We will not diverge too far along that direction.

3 Operations

According to its definition, a stream entity contains a miniature Turing machine whose input symbols are from a subset or a segment of all input symbols to a larger Turing machine, and whose finite control is defined by the stream entity's kernel function. But the overall program defined by stream entities operates as a finite automata (FA), where each state in the FA has a miniature Turing machine inside. Inter-stream-entity relations define precedence constraints among the FA states but do not define connection (i.e. transition) policies, thus the program operates as a FA with ϵ -moves, shown in Figure 3, where the ϵ -moves simply mean undefined transition functions, not important to stream entities³.

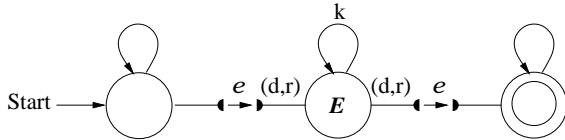


Figure 3: An abstract operation model of a program represented by a finite automata with ϵ -moves in the proposed streaming model nomenclature.

The GeneSC model has all the advantages such as concise foundations and simple semantics of simple operational models without being weak in program clarity, which is guaranteed by detailed designs and implementations of individual stream entities. For

³While reading Snyder's paper [57] after the conception of the GeneSC model, the author could instantly recognize the similarity between his ideal of an operational stream computing model and Snyder's considerations for the candidate type architecture (CTA) abstraction for MIMD parallel machines.

As another note, from our point of view, one major difference between multicore and manycore resembles the difference between shared-memory systems and distributed memory (i.e. networking) systems, where inter-process communication mechanisms set them apart. Our GeneSC model supports both systems.

examples, addition operations in the form of parallel executions of stream entities are supported; concatenation operations are allowed as long as inter-stream-entity relations, if any, are preserved; also allowed are composition operations. With these operations, building large-scale software systems and reusing library components are safe and tractable. The composition capability is even more crucial for hierarchical and scalable computing, which will be discussed in Sections 5.2 and 5.3.

The proposed streaming model preserves asynchronism inherited from the problem domain, but reduces non-determinism to the minimum through explicit concurrency structures in a program⁴. By encapsulating internal computation states, isolating external states, and separating computation from communication, stream entities help reducing data races. Meanwhile, inter-stream-entity relations help reducing general races.

4 Programming

Figure 4 shows a programming framework based on the GeneSC model. At the top level, programmers focus on coding concurrency structures defined by stream entities, which relate to each other according to precedence constraints, if any, that exist in high-level control flows and data flows. Each stream entity has a kernel function, which may consist of none, one or more than one lower-level stream entities. The lowest level kernel functions will call routines in conventional (e.g. non-threaded but thread-safe) programming libraries.

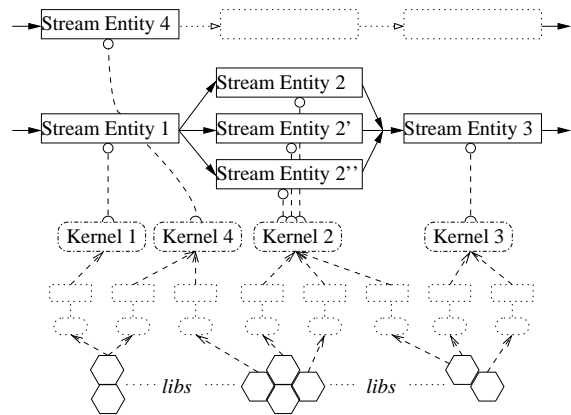


Figure 4: A programming framework based on the GeneSC model.

⁴This model provides the benefit of functional programming in compositional operations but is also intended to yield better performance.

In the above figure, hierarchical structures among stream entities embody compositional operations. Dependence constraints among stream entities decide their runtime execution orders; if there is no dependence constraint between two stream entities, there is no mandatory execution order between them. For example, because stream entity No.4 is independent of others, it may be executed in parallel with stream entity No.1, No.2 or No.3. Programmers see no thread or any low-level parallel execution controls, which are left to compilers, runtime and operating systems to handle. Programmers do have the responsibility to write kernel functions and conventional library routines, if there are no existing ones for reuse.

5 Designs

The GeneSC model provides a highly abstracted, mostly tractable view of concurrent operations. Here are some *examples* as design considerations that support the programming framework discussed in Section 4.

5.1 Language

Stream entity is the foundation of conveying concurrency structures in a program. Though similar concept may already exist in some programming languages, we decide to make it explicit. A stream entity’s API may have the following contents in a new programming-language construct:

```
{
  relations:
  {
    before: {(function_k, [hard/soft]), ...}
    after: {(function_i, [hard/soft]), ...}
  }
  input: <T> x
  output: <T> y
  kernel: function_j
}
```

Inter-stream-entity relations are defined by two sets. Those sets can be empty. Elements in the relation sets are kernel function and relation constraint pairs. A “hard” relation constraint means “this” entity *must finish before* related entities begin, or *must begin after* related entities finish; a “soft” constraint means “this” entity *should finish before* related entities begin, or “this” entity *should begin after* related entities finish. Compositional relations are generated automatically by compilers and are kept in an internal data structure for stream entities.

One major difference between a stream entity and a conventional task in the form of a thread is that

the former makes composition operations easier and safe, whereas the latter harder and unsafe. One major difference between a stream entity and a class object is that the former performs an overall well-defined, close-form function independent of context, whereas the latter does not perform such a function without context, except in the case of a function object or functor [56]. Different from a functor, a stream entity’s internal states are not important.

Stream entity does not imply how its kernel function is implemented, thus does not enforce an imperative language implementation targeting CPU or GPU, or a functional language implementation, or something else. Building stream entities on existing, low-level library routines is possible provided that the library routines do not create POSIX-like threads. In the proposed streaming model, user programs are not encouraged to create threads; library routines should not create threads at all.

The introduction of the stream-entity construct in a programming language is the *first crucial addition* required by our steaming model. In this model, kernel functions define computations while inter-stream-entity relations define precedence constraints in concurrent executions. Inter-stream-entity communications are purposely left out because they are considered as implementation details; both distributed memory methods (e.g. message passing) and shared memory methods are allowed.

At program control level, two more constructs may be desirable: one for data parallelism and one for task parallelism. In the data parallelism case, on the one hand, concurrent executions have a flat structure in term of precedence constraint; some stream entities may have multiple instances at runtime, when each instance takes a portion of a large stream data set. On the other hand, data partition algorithms are often non-trivial. Therefore, a language construct, such as `map` in pMatlab [60], will be useful for conveying the data partition information. Loops had often been used to do the job, but expressing parallel concept in sequential executions reduces program clarity.

In the task parallelism case, sometimes concurrent tasks are divided into groups for completely different functionalities, and have limited synchronization points; sometimes increased security requirements, such as those of a modern web browser [6], demand greater isolation among the tasks. Thus, a task parallelism construct may be needed to define such parallel structures to allow special runtime scheduling and layout treatments.

The research community has been trying to invent new language constructs for the needs of

multithreaded parallel computing, such as in the cases of Cilk/Cilk++ [9, 15] and Galois [32]. The former emphasized the idea of separating designs for computation from concerns for runtime load-balancing and scheduling; the latter emphasized auto-parallelization of sequential programs with new language abstractions. Our language designs align well with those interests, but are not handicapped by the reliance on POSIX-like multithreading model.

5.2 Compiler

The introduction of new language constructs such as that for stream entity has two major impacts on compiler designs.

First, increased power in program analysis and parallelization. Stream entity defines a scope that can be bigger than loop nests, and the scope is finite so is superior than intractable number of procedures that hinder previous practices in interprocedural analyses [1, 21]. In the GeneSC model, a control construct for task parallelism as mentioned in Section 5.1 can guide code generation; a control construct for data parallelism can specify the number of concurrent streams, if programmers provide that information.

Second, added to compiler generated code thus a program’s virtual address space layout, see Figure 5, is a new segment for storing hypergraphs of inter-stream-entity relations, where the hypergraph vertices are stream entities, and the edges are sets of data-flow- or control-flow-connected stream entities; the size of an edge can be one. The hypergraph information is the *second crucial addition* enabled by the proposed stream computing model.

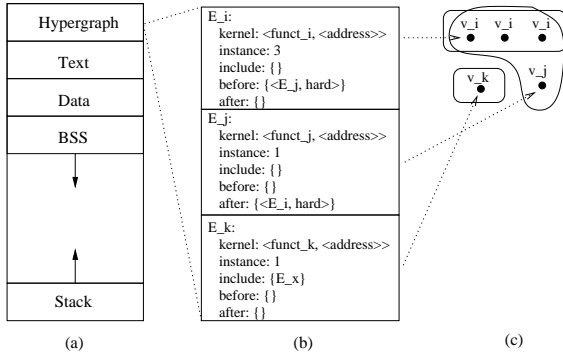


Figure 5: Program virtual address space layout (a), with hypothetical internal structures (b), and drawing of a hypergraph (c).

A hypergraph vertex, i.e. a stream entity, is designed as a non-distributive execution unit, thus

compilers (and later runtime schedulers) are allowed to optimize computations through restructuring (e.g. flattening) the initial hierarchical hypergraphs derived from user programs.

Architectural designs [63, 67] have been proposed to provide much homogeneous programming environment on asymmetric or heterogeneous multicore. Such works make mapping stream entities to asymmetric multicore through compilers practically achievable. Depending on what the underlying processor architecture is, just-in-time (JIT) compilation may be employed to further optimize a program at runtime.

5.3 Operating System

Program speedup technologies that take advantage of parallelism, such as instruction-level pipelines, data-level SIMD, and task-level hyperthreading [22, 28], all have hardware assistants. However, in the multicore case, the up-scaled, parallel processing resource is exposed directly to programmers. Specifically, parallel task creation, mapping and scheduling to multiple cores become programmers’ responsibility, with or without operating system involvement. Such arguable architectural defect or design trade-off may be regarded as the chief culprit for current difficulties in parallel programming on multicore processors. Architectural support for multi-thread (-block, -fiber, -shred, -strand, etc.) programming, like those in [23, 63], would be helpful. Unfortunately, to our best knowledge, no such CPU production exists. To solve that problem, we propose software emulated, e.g. operating-system initiated, task pipelines where a task is a stream entity plus stream data. Such superscalar pipelines are the *third crucial addition* enabled by the proposed streaming model. To illustrate the idea, we use the following shared memory model for more detailed discussions.

First, change how kernel creates and destroys a program’s runtime image. For example, when the `exec` system calls are invoked, a kernel thread, named micro scheduler to be distinguished from the process level scheduler, parses a program’s hypergraphs information, and decides the number of worker threads that form the superscalar pipelines. Work-load information and power management instructions may be available to the micro scheduler. Ideally, it will take those information into consideration when creating worker threads. To reduce context switching⁵ overhead introduced by process level

⁵Context switching is a multiprogramming strategy on single core. In multicore systems, the strategy may need enhancement or extension.

scheduling, the micro scheduler dynamically changes the number of worker threads.

Second, schedule a process' concurrent instruction blocks through work-stealing [9]. Although the micro scheduler speculates inter-stream-entity relations and dynamically control the number of super-scalar pipelines (i.e. worker threads), stream entities are scheduled through work-stealing. Parsed and maybe further optimized hypergraphs information is saved as process context for reuse. Since different operating systems have very different thread models [42, 45], we do not enforce how worker threads are implemented. Figure 6 shows a hypothetical case where the application process has three worker threads, assisted at runtime by a micro scheduler.

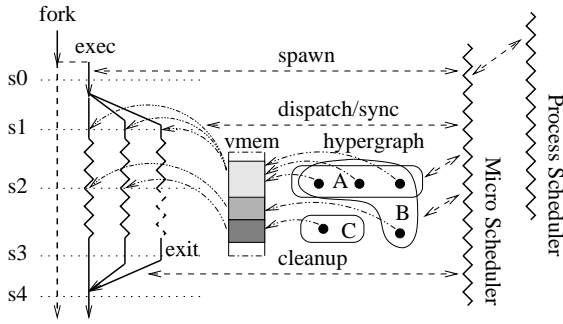


Figure 6: A hypothetical case of operating system kernel initiated threads for stream computing.

Third, monitor virtual memory accesses and manage worker threads synchronizations based on an application's hypergraphs. Virtual memory access control is enforced through shadowing, e.g. locking a memory address so that only one worker thread can write to the address at a specific time; or coloring, i.e. making certain addresses accessible by only one of the worker threads at any time. When race conditions happen, block the contender thread(s) or issue a signal. One immediate question is: how can such fine-grained memory coloring be possible, if modern virtual memory systems are page based? We are considering a dynamic version of memory overlays [39] on top of virtual memory. Memory overlays provide task-level program isolation.

Finally, let operating system kernel cleanup the worker threads at process exit, and output tractable runtime states in case of errors. For example, when system signals make a process abort, hypergraphs information is dumped to a core file. A snapshot of current running stream entities is useful but may not be possible in practice.

We've noticed a recent publication [59] on architecture design built on a concept called chunk execu-

tion, which was defined as a set of sequential instructions executed as one operation unit with processor hardware assistance. Although it is still early for us to assess the hardware design itself, the chunk execution idea reaffirmed us the value of our task-level pipelines for multicore.

6 Application

The modern web browser as a computing platform and a familiar but non-trivial application is used to show the advantages of stream computing.

Besides plain text HTML contents, current and future web browsers are expected to present to end users dynamic and rich media such as video, 3D images, virtual worlds, computer games, semantic web information, etc. Those complex contents require part, if not all, of the computations be done on user computers or smart hand held devices. For each different content format, the browser may run a virtual machine, e.g. a language interpreter, to process the content. There is also the need to isolate content handling tasks inside the browser for privacy and security reasons [6, 54].

Figure 7 shows a simplified pipeline of the rendering engine in a browser. Each of the three functional units, namely parsing, synthesis, and rendering has multiple sub-units of different capabilities, and the sub-units may be composed of different algorithms in implementation.

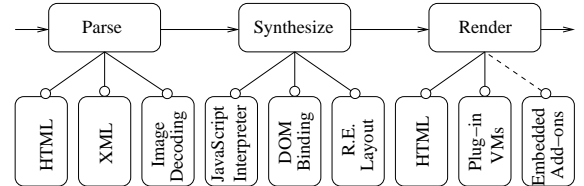


Figure 7: Components in a simplified pipeline of the rendering engine in a modern web browser.

In this case, parallel programming using POSIX threads or OpenMP APIs would be hard. We see at least three problems with the user or compiler-initiated multithreading model: (1) Maintaining scalable number of threads while optimizing parallelization is difficult. (2) Thread creation and destroy overheads may be significant, where threads are temporary objects. (3) Thread isolation is not guaranteed.

By using the proposed streaming model, computations are defined by stream entities. Each component in Figure 7 can be built with stream entities through addition, concatenation, and composi-

tion operations. At runtime, the micro scheduler defined in Section 5.3 keeps a dynamically scalable number of worker threads in a browser, and schedules stream entities as execution units. Isolation is enforced through memory coloring in the browser’s virtual address space.

7 Related Works

All directly related works have been mentioned in the previous sections. What we discuss here are publications that may provide alternative solutions to various aspects of the problems we try to solve, or are noteworthy references to earlier works. We skip publications on virtualization, threads scheduling and thread models, because they are highly specialized topics concerning implementations.

A school of analytical models, represented by the most recent Multi-BSP [62], provides theoretical abstractions that bridge parallel algorithm designs with multicore architectures. One common trait of such models is their emphasis on portable performance. However, such motivation may not ensure engineering success, as there are more imminent concerns, such as indeterminacy, synchronization overhead, and scalability, that highlight the weaknesses of current parallel programming tools. In the Multi-BSP case, the model should be useful in analyzing a given parallel computing system in some multicore architectures, but the model does not provide enough engineering guidance in designing such a system. It is not surprising that the Multi-BSP model’s hierarchically nested component structures resemble the stream entity hypergraphs in our model, but our stream entities have richer engineering implications.

Someone has suggested that the GeneSC model that we promote was a rediscovery of Kahn’s Process Networks [29, 30]. But that’s a misunderstanding, if one can ignore the superficial resemblance in our use of certain terminologies. The simple language Kahn defined and discussed is both synchronous and deterministic, due to the language’s blocking primitives such as “wait” and “send”, which build the language’s FIFO queuing model for communication. The Kahn model suits concurrent tasks, but has little emphasis on high performance computation. We argue that a concurrent computing model should convey intrinsic data and task precedence constraints but should not sacrifice asynchronism when it is possible. The flexibility provided by asynchronism is essential for achieving high-performance through out-of-order execution, thus is needed by a broader range of applications.

In his thesis on StreamIt [58], Thies summarized languages from Kahn’s Process Networks to synchronous dataflow [38] and CSP (Communicating Sequential Processes) [27]. Such languages have a common feature that is the pipe-lining of concurrent task-executions. StreamIt also experimented on the idea of stream graph that allows compiler optimizations. Our main reservation with such languages is their weak power in modeling data parallelism.

Jade [55] represents an important experiment on parallel programming language design. Jade preserves the serial semantics of a program, implicitly exploits task parallelism, and moves data closer to processors. However, if it was designed for a smooth transition from sequential programming to parallel programming, Jade’s dependence on its type system and explicit object-access control still exposes low-level synchronizations to programmers. While arguably such low-level controls provide programming flexibility, not all the details are essential to algorithm design. On the other hand, we might consider that Jade lacks the power for defining concurrent execution structures in a larger program-scope.

Another parallel programming language that has limited adoption in the academia is UPC [61]. UPC is another multithreading language that extends C. The two most noteworthy features of UPC include the partitioned global address space model and the memory consistency model, where the latter is implemented using explicit access controls to shared objects. Again, the weakness of UPC might be its explicit synchronizations exposed to programmers.

Chapel [14] is a complex parallel programming language that supports global level data and control abstractions, architecture-aware locality mapping, object-oriented programming and generic programming. What impressed us most are the language’s global views on data and control, and its ability to map the global views onto data/control affinity abstractions called locales. Though we do not have experience with Chapel programming, by studying its ancestor ZPL [57], we see one motivation that emphasizes both performance and portability on various parallel architectures. For that purpose, ZPL and Chapel may strike for a balance between abstraction and expressiveness. The emphasis of our GeneSC model, however, is (high-level) concurrency structures in problems’ solutions. Therefore, we can rely on extensions to existing languages to achieve concurrency structure abstraction, and delegate performance concerns (e.g. the mapping to parallel architectures) to languages and compilers that implement the kernel functions encapsulated by a stream entity.

We’ve mentioned Cilk++ earlier but consider that it deserves more attention. First, Cilk++ is interesting because of its theoretical foundation and its internal handling of both task-parallelism and data-parallelism. Cilk++ has a provable threading model that can be analyzed using DAG (Directed Acyclic Graph). Conceptually, a Cilk++ strand is like a stream entity of our model, though Cilk++ does not have an explicit construct for strand. Second, Cilk++ uses only two language constructs, namely `cilk_spawn` and `cilk_for`, to define task parallelism and data parallelism at language level, though Cilk++ language constructs have scope restrictions that make C++/Cilk++ mixed-language programming cumbersome [15]. Finally, Cilk++ has restricted but innovative debugging model and hyper-objects support.

In a survey [43] on parallel computing languages for multicore architectures, McCool compared and contrasted existing programming language models and computing paradigms. One shared interest is about stream computing, particularly the “SPMD stream processing model”, implemented on the RapidMind platform [46]. Three of the platform’s major contributions may be summarized as follows. First, at language level, RapidMind adopts a type-system approach inlining with Jade, CUDA and OpenCL; but the type system is much simpler and exposes no synchronization nor access control details. Second, at runtime level, RapidMind schedules and load-balances the executions of concurrent “program objects” through backend (runtime) supports, which identify and parallelize the program objects. Noticeably, a program object has similar purpose as our stream entity but lacks of precedence constraints. Third, the RapidMind platform supports heterogeneous multicores. One needs to be reminded, however, that while it does not relying on conventional threads, RapidMind’s implementation of the SPMD stream processing model limits the platform to applications that show data parallelism.

In [53], the authors proposed a close-to-metal substrate called “lithe”, which uses a primitive named “hart” and its context to carry out application-level threads in multicore processors. Lithe also has an interface layer that allows processor allocation and thread scheduling through runtime systems. The motivation was to control hardware resources subscription and to support parallel library compositions. As an alternative OS-level thread implementation, Lithe has been used to run existing multithreading libraries such as OpenMP.

Bläser in [8] reported a component-based language and operating system that supported concurrent and

structured computing. In that system, a component encapsulates data and computing, which are defined as services to be produced by one component and consumed by another. The emphasis on relations among components is like ours, except that communications among related components use explicit message passing.

Finally, the GeneSC model is for concurrent computing at instruction-block level. Hardware-dependent considerations, such as supports to many-core and heterogeneous multicore, will rely on implementations, particularly those at runtime and operating system levels, such as those in [7, 41, 50].

8 Concluding Remarks

In this paper, we discuss a generalized stream computing model, known as the GeneSC model, for concurrent computing. This model is not about yet another programming technique or a programming paradigm, but an abstract design guidance for parallel programming languages, compilers, runtime and/or operating systems for multicore. We highlight the benefits of the model through operational analysis, and demonstrate its power through an enabled programming framework and example design considerations that support the framework. Specifically, the GeneSC model brings in a non-distributive execution unit called stream entity, and uses it as the foundation for designs at programming language level, compiler level, and operating system level. Besides stream entity, two other crucial additions unlike traditional practices are also introduced: a new hypergraph section in a program’s runtime layout, and operating system kernel initiated threads that form stream-entity/stream-data pipelines to multicore processors. The significance of the GeneSC model is that, by harnessing concurrency information expressed in algorithms and by applying the information to runtime scheduling, the model provides tractable, concurrent operations; and may serve as a natural extension of the von Neumann model for parallel computing architectures.

Our GeneSC model facilitates a high-level abstraction of concurrency in computing, yet does not incur a deep learning curve because the kernel of a stream entity is the encapsulation of a single-/multiple-procedure function that can be implemented in existing sequential languages. By delegating many of the tough issues (e.g. synchronization and task scheduling) in concurrent computing to platform software such as compilers and operating systems, programmers can focus more on analyzing their ap-

plication problems, designing algorithms, and defining concurrency structures of the solutions—a shift from machine-oriented programming to application-oriented programming; at the meantime, sequential programming develops smoothly into concurrent programming.

Acknowledgments. The author learned of the existence of hypergraphs through professor Vitaly Voloshin of the Troy State University, USA. An anonymous reader of the first draft of this paper brought up Kahn’s works, which led to the comparison between Kahn’s model and ours. The author once questioned professors Peter J. Denning and Jack B. Dennis about their discussion of determinate computation in an article published in CACM, Vol. 53, No. 6; the two professors’ kind replies further clarified this author’s use of certain terminologies.

References

- [1] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures, A Dependence-Based Approach*, Morgan Kaufmann Publishers, 2002.
- [2] K. Asanovic, R. Bodik, B. C. Catanzaro, et al, *The Landscape of Parallel Computing Research: A View from Berkeley*, Technical Report No. UCB/EECS-2006-183, Dec. 2006.
- [3] J. Backus, Can Programming Be Liberated From the von Neumann Style? A Functional Style and Its Algebra of Programs, *OCCAM*, Vol. 21, No. 8, 1978, pages 613-641.
- [4] S. Bhatt, M. Chen, C. Lin, and P. Liu, Abstractions for Parallel N-body Simulations, In *Proceedings of Scalable High Performance Computing Conference*, IEEE Press, April 1992, pages 38-45.
- [5] J. Barnes and P. Hut, A Hierarchical $O(N \log N)$ Force-Calculation Algorithm, *Nature*, Vol. 324, No. 4, December 1986, pages 446-449.
- [6] A. Barth, C. Jackson, C. Reis, and Google Chrome Team, The Security Architecture of the Chromium Browser, <http://seclab.stanford.edu/websec/chromium>, 2008.
- [7] A. Baumann, P. Barham, P. Dagand, et al., The Multikernel: A New OS Architecture for Scalable Multicore Systems, In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM Press, Oct. 2009.
- [8] L. Bläser, A High-Performance Operating System for Structured Concurrent Programs, In *Proceedings of the 4th workshop on Programming languages and operating systems*, Oct. 2007.
- [9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, et al., Cilk: An Efficient Multithreaded Runtime System, *Technical Report TM-548*, Massachusetts Institute of Technology, 1996.
- [10] H. Boehm, Threads Cannot Be Implemented As a Library, In *Proceedings of the 2005 ACM SIGPLAN Conference on Parallel Language Design and Implementation*, ACM Press, Jun., 2005, pages 261-268.
- [11] H. Boehm, Transactional Memory Should Be an Implementation Technique, Not a Programming Interface, In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (Hot-Par’09)*, March, 2009.
- [12] I. Buck, *Stream Computing on Graphics Hardware*, Ph.D. thesis, Stanford University, 2006.
- [13] D. R. Butenhof, *Programming with POSIX Threads*, Addison-Wesley Publishing Company, 1997.
- [14] *Chapel Language Specification 0.782*, Cray Inc., 2009.
- [15] Cilk Arts, *Cilk++ Programmer’s Guide*, Revision 58, Cilk Arts, Inc., March 16, 2009.
- [16] R. Cleaveland, S. A. Smolka, et al, Strategic Directions in Concurrency Research, *ACM Computing Surveys*, Vol. 28, No. 4, December 1996, pages 607-625.
- [17] W. Dally, P. Hanrahan, M. Erez, et al., Merimac: Supercomputing with Streams, In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ACM Press, Nov., 2003.
- [18] R. Gerber, *The Software Optimization Cookbook, High-performance Recipe for the Intel Architecture*, Intel Press, 2002.
- [19] R. Greenlaw, H. J. Hoover, and W. L. Ruzzo, *Limits to Parallel Computation, P-Completeness Theory*, Oxford University Press, 1995.
- [20] J. Gummaraju, M. Erez, J. Coburn, et al., Architecture Support for the Stream Execution Model on General-Purpose Processors, In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, IEEE Computer Society Press, Sept., 2007, pages 3-12.

- [21] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, et al., Interprocedural Parallelization Analysis in SUIF, *ACM Transactions on Programming Languages and Systems*, Vol. 27, No. 4 (July), 2005, pages 662-731.
- [22] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th Edition, Morgan Kaufmann Publishers, 2006.
- [23] R. A. Hankins, G. N. Chinya, J. D. Collins, et al., Multiple Instruction Stream Processor, In *Proceedings of the 33rd International Symposium on Computer Architecture (ISCA'06)*, IEEE Computer Society Press, 2006.
- [24] M. Herlihy and J. E. B. Moss, Transactional Memory: Architectural Support for Lock-Free Data Structures, In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, IEEE Computer Society Press, 1993, pages 289-300.
- [25] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers, 2008.
- [26] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley Publishing Company, 1979.
- [27] C.A.R. Hoare, *Communicating Sequential Processes*, Prentice Hall International, 1985. Also freely available on the Internet since 2004.
- [28] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Order No. 248966-014, Intel Corporation, Nov., 2006.
- [29] G. Kahn, The Semantics of a Simple Language for Parallel Programming, *Information Processing 74*, North-Holland Publishing Co., 1974, pages 471-475.
- [30] G. Kahn and D. B. MacQueen, Coroutines and Networks of Parallel Processes, *Information Processing 77*, North-Holland Publishing Co., 1977, pages 993-998.
- [31] Khronos Group, OpenCL Parallel Computing for Heterogeneous Devices, <http://www.khronos.org/opencv1>, 2009.
- [32] M. Kulkarni, K. Pingali, B. Walter, et al., Optimistic Parallelism Requires Abstractions, *CACM*, Vol. 52, No. 9, 2009, pages 89-97.
- [33] F. Labonte, P. Mattson, W. Thies, and I. Buck, The Stream Virtual Machine, In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, IEEE Computer Society Press, 2004.
- [34] Larrabee Microarchitecture, <http://www.intel.com/technology/visual/microarch.htm>, Jan., 2010.
- [35] J. Larus and C. Kozyrakis, Transactional Memory: Is TM the Answer for Improving Parallel Programming?, *CACM*, Vol. 51, No. 7, 2008, pages 80-88.
- [36] E. A. Lee, The Problem with Threads, *IEEE Computer*, Vol. 39, No. 5, May, 2006, pages 33-42.
- [37] E. A. Lee, Computing Needs Time, *CACM*, Vol. 52, No. 5, May 2009, pages 70-79.
- [38] E. A. Lee and D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, *IEEE Transactions on Computers*, Vol. 36, No. 1, 1987, pages 24-35.
- [39] J. R. Levine, *Linkers & Loaders*, Morgan Kaufmann Publishers, 2000.
- [40] S. Liao, Z. Du, G. Wu, and G. Lueh, Data and Computation Transformations for Brook Streaming Applications on Multiprocessors, In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'06)*, IEEE Computer Society Press, March 2006.
- [41] R. Liu, K. Klues, S. Bird, et al., Tessellation: Space-Time Partitioning in a Manycore Client OS, In *Proceedings of the First USENIX Workshop on Hot Topics in Parallelism (HotPar'09)*, March, 2009.
- [42] R. Love, *Linux Kernel Development*, 2nd Edition, Novell Press, 2005.
- [43] M. D. McCool, Scalable Programming Models for Massively Multicore Processors, *Proceedings of the IEEE*, Vol. 96, No. 5, May 2008.
- [44] P. S. McCormick, J. Inman, J. P. Ahrens, et al., Scout: A Hardware-Accelerated System for Quantitatively Driven Visualization and Analysis, In *Proceedings of the Conference on Visualization '04*, ACM Press, 2004.

- [45] R. McDougall and J. Mauro, *Solaris Internals, Solaris 10 and OpenSolaris Kernel Architecture*, 2nd Edition, Sun Microsystems Press, 2006.
- [46] M. Monteyne, RapidMind Multi-Core Development Platform, *RapidMind White Paper*, RapidMind™, Feb. 2008.
- [47] R. H. B. Netzer and B. P. Miller, What Are Race Conditions? Some Issues and Formalizations, *ACM Letters on Programming Languages and Systems*, Vol. 1, No. 1, March 1992, pages 74-88.
- [48] News on Larrabee's delay again, <http://arstechnica.com/hardware/news/2009/12/intels-larrabee-gpu-put-on-ice-more-news-to-come-in-2010>, Dec. 04, 2009.
- [49] J. Nickolls, I. Buck, M. Garland and K. Skadron, Scalable Parallel Programming with CUDA, *Queue*, Vol. 6, No. 2, 2008, ACM Press, pages 40-53.
- [50] E. B. Nightingale, O. Hodson, R. McIlroy, et al., Helios: Heterogeneous Multiprocessing with Satellite Kernels, In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM Press, Oct. 2009.
- [51] NVIDIA CUDA Compute Unified Device Architecture Programming Guide, Version 1.1, NVIDIA Corporation, Nov., 2007.
- [52] OpenMP Application Program Interface, Version 3.0, <http://openmp.org>, May, 2008.
- [53] H. Pan, B. Hindman, and K. Asanović, Composing Parallel Software Efficiently with Lithe, In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, June, 2010.
- [54] C. Reis and S. D. Gribble, Isolating Web Programs in Modern Browser Architectures, In *Proceedings of the 4th ACM European conference on Computer systems (EuroSys'09)*, ACM Press, 2009, pages 219-231.
- [55] M. C. Rinard and M. S. Lam, The Design, Implementation, and Evaluation of Jade, *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 3, May 1998.
- [56] B. Stroustrup, *The C++ Programming Language*, Special Edition, Addison-Wesley Publishing Company, 2001.
- [57] L. Snyder, The Design and Development of ZPL, In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ACM Press, 2007.
- [58] W. Thies, *Language and Compiler Support for Stream Programs*, PhD Thesis, Massachusetts Institute of Technology, 2009.
- [59] J. Torrellas, L. Ceze, J. Tuck, et al., The Bulk Multicore Architecture for Improved Programmability, *CACM*, Vol. 52, No. 12, 2009, pages 58-65.
- [60] N. Travinin and J. Kepner, *pMatlab Parallel Matlab Library*, Lincoln Lab, MIT, 2006.
- [61] UPC Consortium, *UPC Language Specifications*, Version 1.2, May 31, 2005.
- [62] L. G. Valiant, A Bridging Model for Multi-Core Computing, *Journal of Computer and System Sciences*, Vol. 77, Issue 1, Jan. 2011, pages 154-166.
- [63] P. H. Wang, J. D. Collins, G. N. Chinya, et al., EXOCHI: Architecture and Programming Environment for A Heterogeneous Multithreaded System, In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, ACM Press, June 2007.
- [64] Y. Wang, R. M. Hyatt, and B. R. Bryant, "Architectural considerations with distributed computing", In *Proceedings of the 2nd International Conference on Enterprise Information Systems (ICEIS 2000)*, July 2000, pages 535-536.
- [65] M. S. Warren and J. K. Salmon, Astrophysical N-body Simulation Using Hierarchical Tree Data Structures, In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, IEEE process, 1992, pp.570-576.
- [66] B. Wilkinson and M. Allen, *Parallel Programming, Techniques and Applications Using Networked Workstations and Parallel Computers*, Prentice Hall, Inc., 1999.
- [67] H. Wong, A. Bracy, E. Schuchman, et al., Pangaea: A Tightly-Coupled IA32 Heterogeneous Chip Multiprocessor, In *Proceedings of Parallel Architectures and Compilation Techniques (PACT 08)*, 2008.
- [68] *Working Draft, Standard for Programming Language C++*, Document No. N2798, ISO/IEC, Oct., 2008.